AD-A067 892 ARIZONA UNIV TUCSON DEPT OF CHEMISTRY F/G 9/2
ADVANCED SOFTWARE CONCEPTS FOR EMPLOYING MICROCOMPUTERS IN THE --ETC(U)
APR 79 S B TILDEN, M B DENTON N00014-75-C-0513
UNCLASSIFIED TR-18 NL

1 OF 1
AD A067892

END
DATE FILMED
6 -79
DDC

OFFICE OF NAVAL RESEARCH
Contract N00014-75-C-0513
Task No. NR 051-549
Technical Report No. 18

ADVANCED SOFTWARE CONCEPTS FOR EMPLOYING MICROCOMPUTERS IN THE LABORATORY

Scott B. Tilden and M. Bonner Denton

Department of Chemistry
University of Arizona
Tucson, Arizona 85721

Prepared for Publication

in

Journal of Automatic Chemistry

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-18	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): ADVANCED SOFTWARE CONCEPTS FOR EMPLOYING MICROCOMPUTERS IN THE LABORATORY	5. TYPE OF REPORT & PERIOD COVERED: INTERIM rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Scott B. Tilden and M. Bonner Denton	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0513
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Chemistry, University of Arizona, Tucson, AZ 85721 — 26p.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 051-549
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, Virginia 22217	12. REPORT DATE: April 1979
	13. NUMBER OF PAGES: 9
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

CONVERS
Interpretive Compiler
Computer Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Current trends in the computerization of specialized custom chemical instrumentation indicate the increasing utilization of dedicated microprocessors. Conventional software techniques often possess serious limitations in regard to initial development effort, execution speed, flexibility, and/or hardware required. A high-level "interpretive compiler" software package CONVERS is described which offers numerous advantages compared to conventional approaches including high speed operation, high level I/O, language flexibility, superior memory efficiency and a variety of other desirable characteristics.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

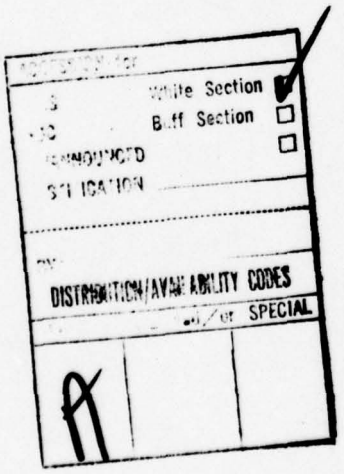
S/N 0102-014-6601

033 860

ADVANCED SOFTWARE CONCEPTS FOR EMPLOYING MICROCOMPUTERS IN THE LABORATORY

Scott B. Tilden and M. Bonner Denton

Department of Chemistry
University of Arizona
Tucson, Arizona 85721



ABSTRACT

Current trends in the computerization of specialized custom chemical instrumentation indicate the increasing utilization of dedicated microprocessors. Conventional software techniques often possess serious limitations in regard to initial development effort, execution speed, flexibility, and/or hardware required. A high-level "interpretive compiler" software package CONVERS is described which offers numerous advantages compared to conventional approaches including high speed operation, high level I/O, language flexibility, superior memory efficiency and a variety of other desirable characteristics.

Advanced Software Concepts for Employing Microcomputers in the Laboratory

by

Scott B. Tilden and M. Bonner Denton
Department of Chemistry
University of Arizona
Tucson, Arizona 85721

While a proliferation of commercial chemical instrumentation is appearing today employing microprocessors for a variety of control and data reduction applications, the great potential of microprocessors has not been exploited extensively for individual custom applications. The primary reason for this phenomenon is altogether too clear--microprocessor software is either difficult to develop or inefficient in memory requirements and speed. This problem is even more important in situations requiring constant software modification. Initially, most instrument manufacturers utilized cross assemblers supported on large "number cruncher" computers to generate the required machine code binary program. More recently, the trend has been toward the use of a "developmental system" (at a cost comparable to a moderate minicomputer--the authors use the term "mini" in contrast to "micro" reluctantly because of the ever increasing overlap in computing capability) to write and debug assembly level programs which are subsequently converted to binary and incorporated into an instrument in the form of "read only memory" (ROM).

While this approach has proven cost effective for high volume mass produced applications, it possesses serious limitations for system updates and custom applications. Additionally, the ability to program efficiently at the assembly level is a talent requiring a significant expenditure of time to develop.

During the past several years, a virtual deluge of sophisticated, flexible, high performance computer hardware has been introduced primarily aimed at a rapidly growing "hobbyist" market. Manufacturers quickly realized that to sell the public hardware, some form of reasonably high level software must be made available. A variety of BASIC interpreters, ranging from rather "dumb" to "quite intelligent" have since evolved. The more intelligent BASIC interpreters have several highly attractive attributes for "hobbyist" applications. The language is both easy to master and conversational. Error and caution messages are provided as aids during programming.

Why not apply the "hobbyist" technology toward the implementation of custom laboratory systems? Many investigators have and, no doubt, many more will take this approach. However, BASIC interpreters possess serious limitations in terms of system speed, flexibility, and input/output (I/O) capabilities. In BASIC, each command must first be interpreted and then executed (see Figure 1). In many cases, the interpretation process takes much more time than the actual execution. This problem is compounded by the fact that commands interpreted in the past must be re-interpreted each time they are used causing iterative programs to be very slow. While speed is often not a serious limitation

in playing computer games, laboratory applications requiring high speed data acquisition and/or data manipulation are common. Additionally, the more intelligent BASICs make very inefficient use of memory often requiring a minimum of 12 or 16 K bytes (twelve or sixteen thousand eight bit words).

In contrast to interpreters, high level compilers, such as FORTRAN, offer a much faster "run time" execution speed. This is accomplished through generation of the required machine code during a series of programming operations. Compilers using FORTRAN, which are designed to run on many minicomputers and some micros, often first transform user symbolic source code into assembly code. An assembler program, subsequently, transforms this into the required machine code. This ready-to-run machine code is often loaded along with a run time package which executes in the manner shown in Figure 2. While this approach greatly improves execution speed, the need for loading several different software routines increases the "hassle" associated with editing and debugging. Thus, this makes some form of mass memory, such as a disk or magnetic tape, almost mandatory. Additionally, I/O algorithms generally must be implemented in assembly level code!

One obvious question immediately arises--why not incorporate the most desirable characteristics of both interpreters and compilers into a single language? Additionally, due to the unique requirements found in many applications, why not allow the programmer additional flexibility by providing him with the ability to actually develop his own individual modifications and additions to the language itself? Other desirable

features would include high memory efficiency, high level I/O programming, ease of understanding the language's "inter-workings" and the ability to be transferred from one CPU to another with minimum effort.

During the past two years, a different approach to software has been taking place at the University of Arizona referred to as an "Interpretive Compiler" called CONVERS. This package, which is conceptually similar to the FORTH language currently being used in several minicomputer-astronomical applications (1), is able to provide many of the desirable features found in both interpreters and compilers by separating the compile and execute states (as a compiler does) while maintaining a resident user interactive and conversation executive which oversees system operation. The ability to realize such advanced software capabilities in a very modest amount of memory (less than 4 K bytes on an 8080 based micro) is the direct result of exploiting threaded code programming techniques (see Figure 3). The approach involves highly efficient use of simple macroinstructions to build more complex subroutines which are recombined with additional macroinstructions to form super subroutines. This process of combining previously defined modules to form ever increasingly sophisticated routines for performing the task at hand is the essence of threaded code programming. When initially loaded and running, CONVERS acts much like an interpreter, i.e. it is conversational, ready to either execute a previously programmed algorithm or accept a new one. However, in contrast to BASIC, when a new program is being entered under CONVERS, it is immediately transformed into binary machine code or to the binary starting addresses of other previously entered

and compiled machine code programs. During this process, the operator is kept informed of the status of the program by a series of error and diagnostic messages. When the new program has been completed, it is entered in a program library or dictionary, which is constantly building up from low memory (see Figure 4). If the operator now wants to execute this program, he can request it from his terminal. A dictionary search will begin at the last entry and progress until the requested program is located. Once located, the requested program will run in its entirety without need for any additional dictionary searches. For example, let us assume an algorithm, called ACQUIRE, has been programmed to take data from some hypothetical experimental system. When ACQUIRE is requested from the terminal, a dictionary search is initiated. The program named ACQUIRE (see Figure 5), once located, contains the starting addresses of a series of previously defined modules which implement the various steps necessary to perform the desired experiment. For example, the module SCAN which might be intended to scan a monochromator's wavelength in some desired manner has been previously defined and tested. This ability to easily test each module separately and then efficiently combine a series of modules to perform a more complex function, test this function, and then employ it in a vastly more complex function, etc., i.e. testing each step as the threaded code is made increasingly complex, is a major factor contributing to the speed with which software can be developed using CONVERS. Use of a software stack also contributes toward improved memory efficiency and simplified programming.

The stack is an area of memory set aside to handle parameters, data numbers, etc. One of the primary advantages of the stack is that entries can leave temporary parameters on the stack without having to assign specific memory locations to store them. This not only can save considerable memory, but also allows programs to be easily relocatable since one algorithm need only know that a previous routine left so many words of data, etc. on the stack. It need now know where the previous routine is nor even where the stack is located. A series of stack handling routines, which should appear quite familiar to many small calculator users, provide an array of capabilities, including the ability to "PUSH" a number on the stack, "POP" it off, duplicate it, "SWAP" the top two numbers, locate a number some distance into the stack, and copy it on top of the stack, etc. Additionally, a variety of logic functions familiar to the minicomputer user are provided including OR, AND, shift left, shift right, greater than, less than, etc., etc.

Input/output (I/O) is normally accomplished using the stack in conjunction with the "INDEVICE" or "OUTDEVICE" commands. For example, to take data from a device located at I/O, port 7, the number seven is "pushed" onto the stack and INDEVICE is called. INDEVICE "pops" the top number from the stack ("7"), goes to this I/O port, takes in a number and "pushes" the number on the stack. OUTDEVICE functions in a similiar manner, requiring the number to be sent to the desired device to be "pushed" onto the stack followed by the device's I/O port address. Hence, to send the number 131 to device 11, the number 131 is pushed on the stack followed by 11 and than OUTDEVICE. This "pops" the top number

(11) from the stack, uses it as the output port and then sends the number 131 to that location.

To appreciate the ease with which real programs can be written, a few examples will be considered. A trivial program, called SOUND, which rings the terminal bell three times, might be written

```
: SOUND  BELL  BELL  BELL  ;
```

The colon denotes changing from EXECUTE to COMPILE mode. After typing the name of the new routine, in this case to be called "SOUND", typing the name of the earlier defined routine ('BELL' - a previously defined simple program to ring the terminal bell) initiates a dictionary search to locate this routine's starting address which subsequentially is entered three times. The resulting 'SOUND' routine contains machine code calls to the 'BELL' routine which, itself, is composed of machine code. Of course, 'SOUND' could also have been defined using a DO-LOOP, i.e.

```
:  SOUND  3  1  DO  BELL  LOOP  ;
```

where the numbers three and one set the upper and lower indices. If it were desirable to change the actual number of bell rings from some other program, this value could be defined as a VARIABLE--let's call it NOISE.

```
3 VARIABLE  NOISE
```

In this case, the number three is first pushed on the stack, VARIABLE transfers the top number on the stack (the three) to a dictionary location named NOISE. If SOUND were now defined as:

```
: SOUND  NOISE  @  1  DO  BELL  LOOP  ;
```

the bell would again ring three times. In this case, when the word

NOISE is encountered, its address is pushed on the stack, the @ is a simple program which goes to the address indicated on the top of the stack (that of NOISE) and replaces it with the actual value located at that address (the number three). At any future time, the value of a VARIABLE can be changed by "pushing" the new value onto the stack, followed by the address of the variable to be changed, generated by its name and an exclamation mark. To change NOISE to 5,

```
5 NOISE !
```

a number five is pushed onto the stack, NOISE pushes its address on the stack, and ! goes to the address indicated by the "top" number on the stack and deposits the next number. Now sound would ring the terminal bell five times.

A much less trivial program which could be written to scan and take data from a monochromator equipped with a DENCO SM2A stepper motor controller (1) (the SM2A takes a parallel number as an address, sends one of two stepper motors to this location and outputs an arrival flag when the address is reached) is given in Figure 6. Assume that the experimental system is configured so the SM2A is at I/O, port 5 and an analog to digital converter to acquire data is at I/O, port 7. Let us assume that, initially, a scan is designed from a starting stepper motor location of 2000 to a final location of 5000, taking data every 20 steps.

While the code might look a little strange at first, it quickly becomes very easy to work with. The SCAN program of Figure 6 could be combined with other modules as shown in Figure 5 to perform some more complex experimental function. Each module of the program can be

easily tried out to ensure that it is operational before proceeding with the next.

Presently, CONVERS is being used in the authors' laboratories for a variety of spectrochemical investigations, including laser excited optoacoustic spectroscopy (Figure 7) and inductively coupled plasma optical emission spectroscopy (Figure 8). Rather complex interactive control and data acquisition programs have been easily implemented. Memory requirements and operating speed have been found to be far superior to conventional approaches. Additionally, new system users have encountered almost no difficulty in utilizing previously developed software even when documentation was vague.

The authors hope that this short introduction to only a few of the concepts employed in CONVERS will generate interest in its capabilities. A much more complete discussion is available in the form of a user's manual (3) available from the authors.

The development of the CONVERS system was partially supported by the Office of Naval Research and a Alfred P. Sloan Foundation Research Fellowship to M. Bonner Denton.

(1) C. Moore, Astron. Astrophys. Suppl., 15 (1974) 497.

(2) M. B. Denton, J. D. Mack, M. W. Routh and D. B. Swartz, American Laboratory, 8, 69 (1976).

(3) CONVERS: AN INTERPRETIVE COMPILER, developed by Scott B. Tilden and M. Bonner Denton, Department of Chemistry, University of Arizona, Tucson, Arizona 85721.

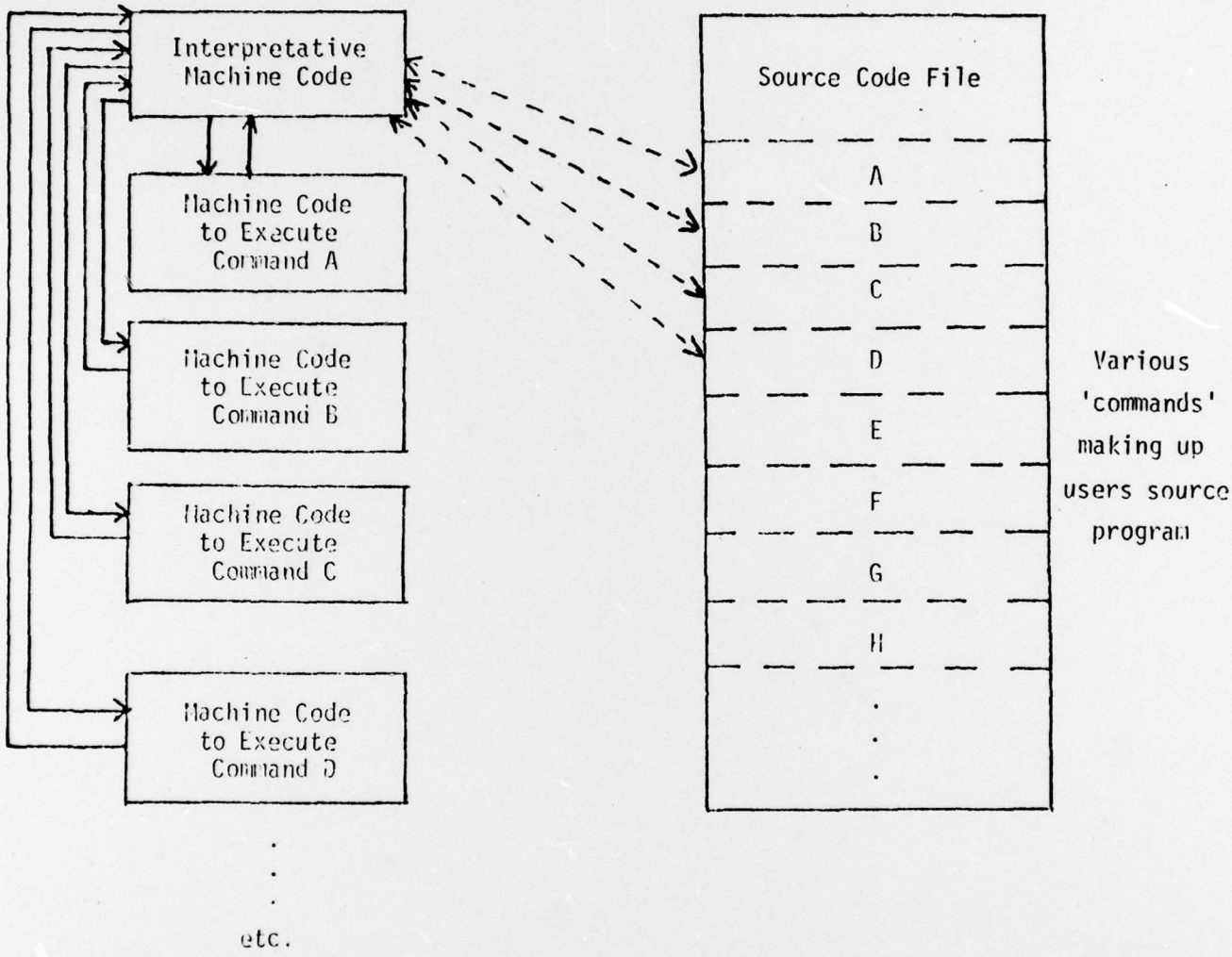


Figure 1: The interpretative cycle of common types of languages such as BASIC. After examining each command in the source file, the interpreter searches for and branches to the corresponding block of machine code; thus, program execution always remains within the interpreter.

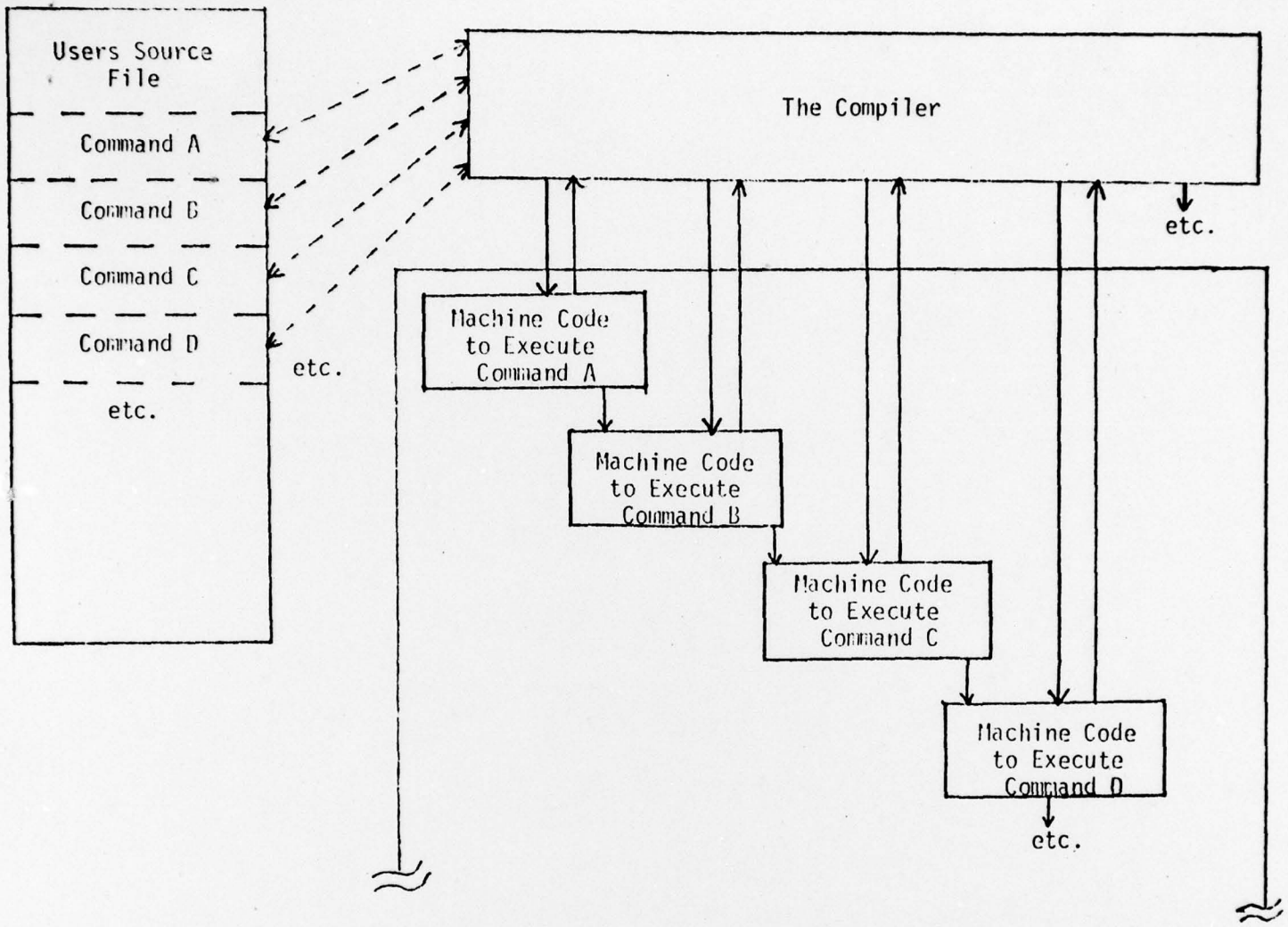


Figure 2: Note that the compiler transforms each source "command" into executable machine code. This code will, subsequently, be loaded and executed independently of the compiler.

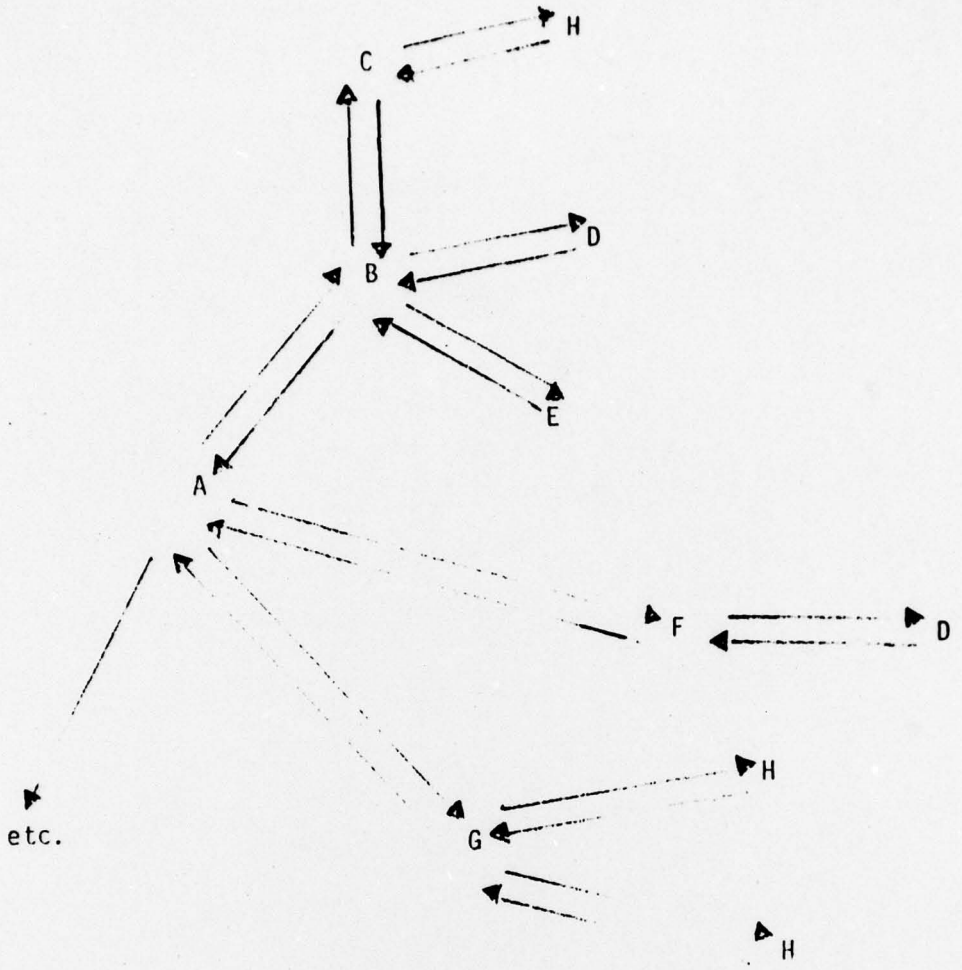


Figure 3. The 'threaded' code approach used in CONVERS. Note that the flow of logic threads its way in a very non-linear fashion.

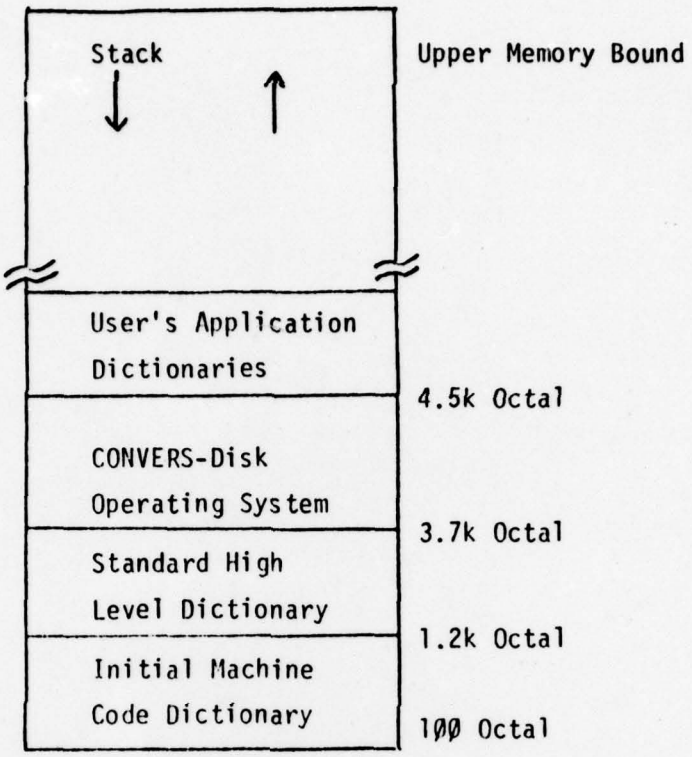


Figure 4. Memory map of the CONVERS dictionary.

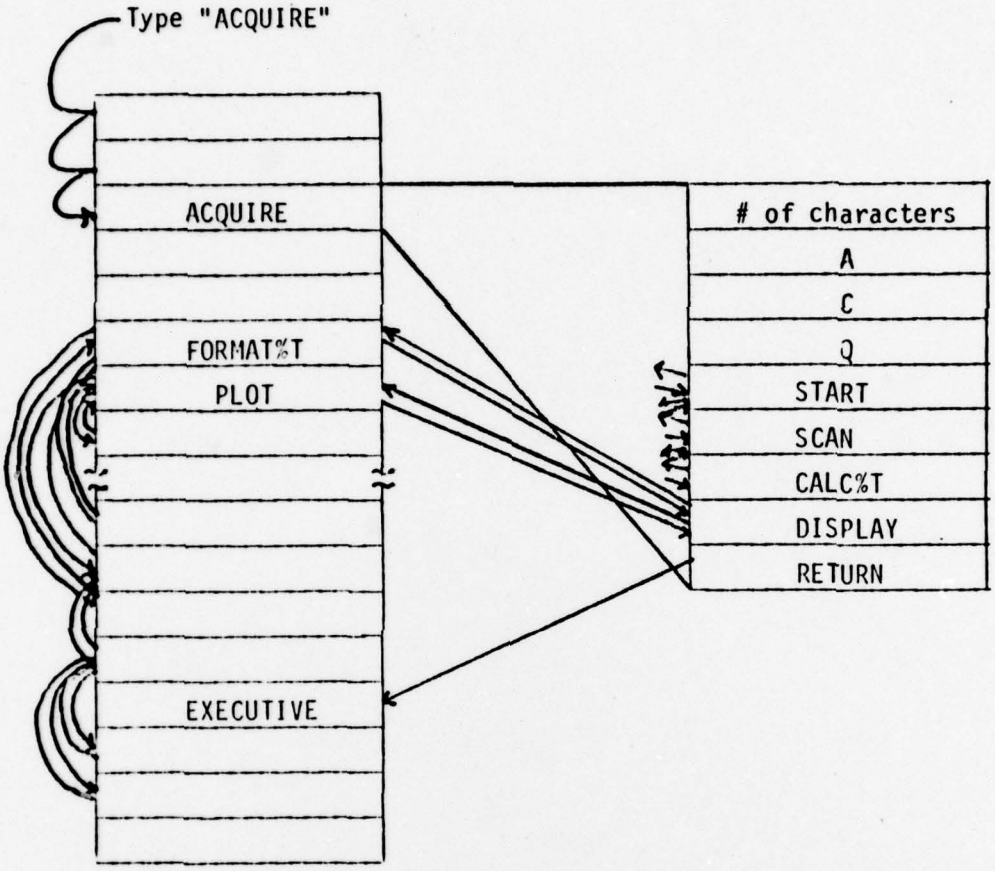


Figure 5. If a previously defined program name (ACQUIRE) is entered when in EXECUTE mode, a dictionary search takes place locating the ACQUIRE entry. Once found, this entry contains all the required machine code and/or calls to addresses of other previously compiled machine code modules to completely execute the desired function.

```
2000  VARIABLE  START
```
Defines a variable called start to be 2000. This will be starting location of scan.

```
5000  VARIABLE  STOP
```
Defines end of scan location.

```
  20  VARIABLE  INCREMENT
```
Defines distance between data points.

```
   0  VARIABLE  LOCATION
```
Defines a variable called LOCATION where next address will be kept.

```
: A/D7  7  INDEVICE  ;
```
Colon puts system in Compile Mode A/D7 will be the name of module which when called will cause a seven to be pushed onto the stack, INDEVICE will POP it back off and use it as a device address to go to and take in a data point and push the data onto the stack.

```
: INITIALIZE  START  @  LOCATION  !  ;
```
Defines a module called "INITIALIZE" "START" puts its address on the stack "@" replaces the address with the value at that address, "LOCATION" puts its address on the stack, "!" goes to the address specified by the top number on the stack and deposits the second number--net result value at "START" is put into "LOCATION".

```
: STEP  LOCATION  @  INCREMENT  @  +  LOCATION  !  ;
```
Defines "STEP" to take values from "LOCATION" and "INCREMENT" add them together and put result back into "LOCATION" i.e., "LOCATION" puts its address on stack, "@" replaces top value on stack with the number stored at that address, "INCREMENT @" gets value at "INCREMENT" and puts it on stack "+" adds top two stack numbers and pushes result on stack, "LOCATION" puts its address on stack and "!" goes to address specified by top number on stack and deposits second number.

```
: DELAY  BEGIN-HERE  5  INDEVICE  10  AND  IF  A/D7  ELSE  BEGIN  THEN  ;
```
This module takes in a number from the stepper motor controller (assume device 5) which pushes it on the stack, pushes the value 10 on the stack does a logical AND to see if the controllers flag is set, if this is true, the A/D7 module will be called to input data, if the flag is not set, the program is returned to "BEGIN-HERE". Therefore, the "DELAY" module is a loop waiting for the stepper motor to arrive at its new location followed by a call to the A/D7 data acquisition module.

```
: MOVE  LOCATION  @  5  OUTDEVICE  STEP  DELAY  ;
```
A module called "MOVE" is defined to get the value stored at "LOCATION", push the device code of the stepper motor controller (5) onto the stack, perform an outdevice (which uses the top stack number as a I/O port address to send the next value to i.e. the value from "LOCATION@") call the "STEP" module (which increments "LOCATION" by the "INCREMENT" value and finally calls "DELAY" which waits for a flag from the stepper motor controller signaling arrival at the desired address and then takes a data point.

```
: SCAN  INITIALIZE  BEGIN-HERE  MOVE  LOCATION  @  STOP  @  >  IF  END  ELSE
BEGIN  THEN  ;
```
The final module called "SCAN" calls the "INITIALIZE" module (which sets LOCATION to the START location for the scan), calls "MOVE" (which sends this value to the motor controller increments the value stored in LOCATION by "INCREMENT" and calls "DELAY" which waits for the motor to arrive and then takes a data point). Next, the incremented value from "LOCATION" is placed on the stack ("LOCATION@") followed by the STOP value ("STOP @") the two are compared to ">" to see if the incremented value at "LOCATION" is larger then the "STOP" value if it is the program end, if not, it repeats starting at "BEGIN-HERE".

NOTE: While many values have been pushed on the stack, only the data will remain, since each time a value is used, it is popped (removed) off of the stack.

If a different spectral region is to be scanned, i.e., from 3000 to 6500 with 10 increments merely change the variables by

```
3000  START      !
6500  STOP       !
  10  INCREMENT  !
```

and type SCAN. Now the system will scan from 3000 to 6500 taking data every 10 steps.

Figure 6. On first inspection, this program to scan wavelength between two easily changed limits and acquire data looks somewhat strange, but by applying the simple rules given in the text examples, it can be readily understood. Remember a number or name pushes the number of address occupied by the name on the stack. The symbol, @, pops the top number from the stack, uses it as an address from which to obtain a number and pushes that number on the stack.

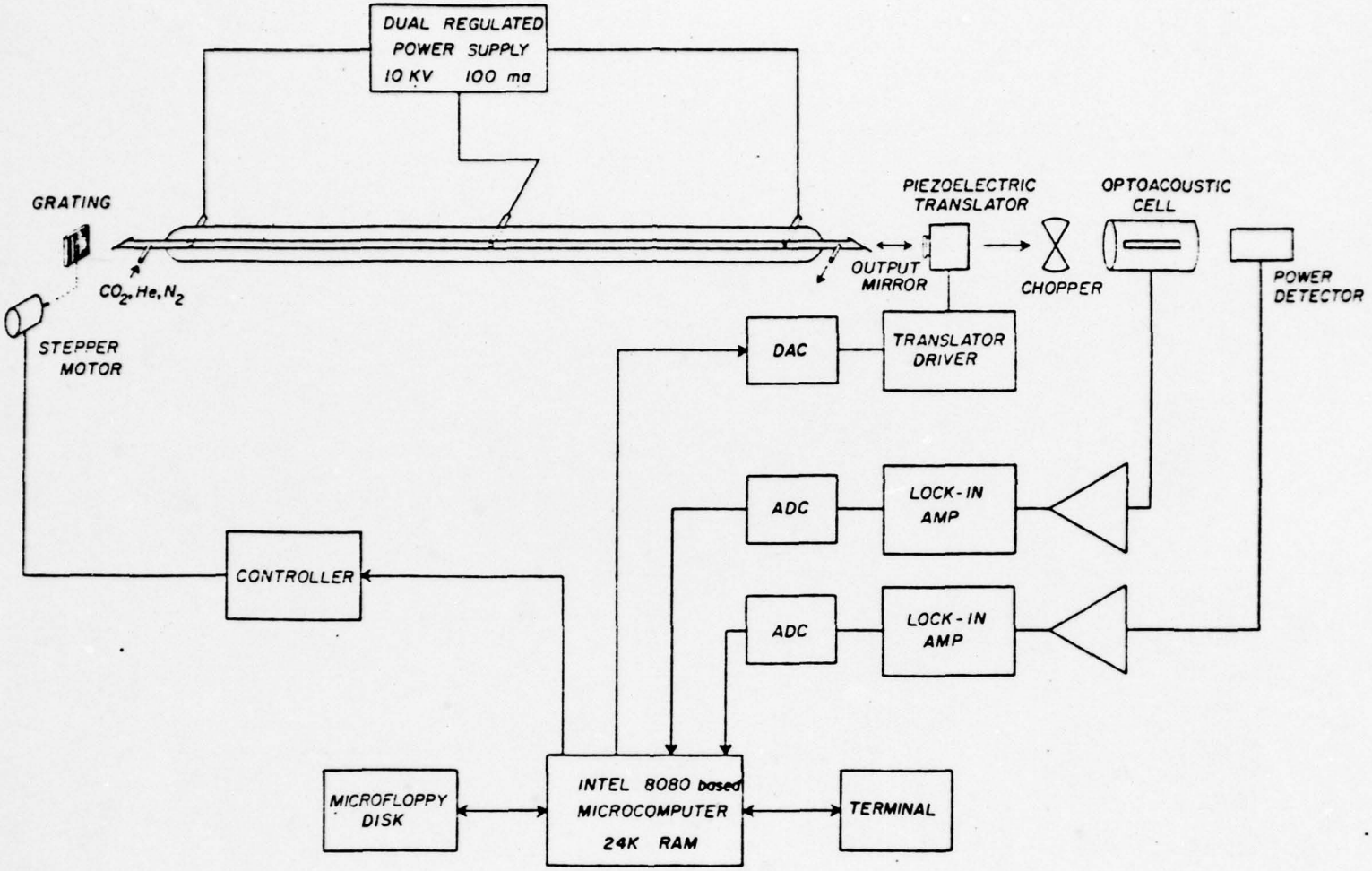


Figure 7. The optoacoustic experiment in which a microcomputer is used to control laser wavelength and to monitor laser power and optoacoustic signal.

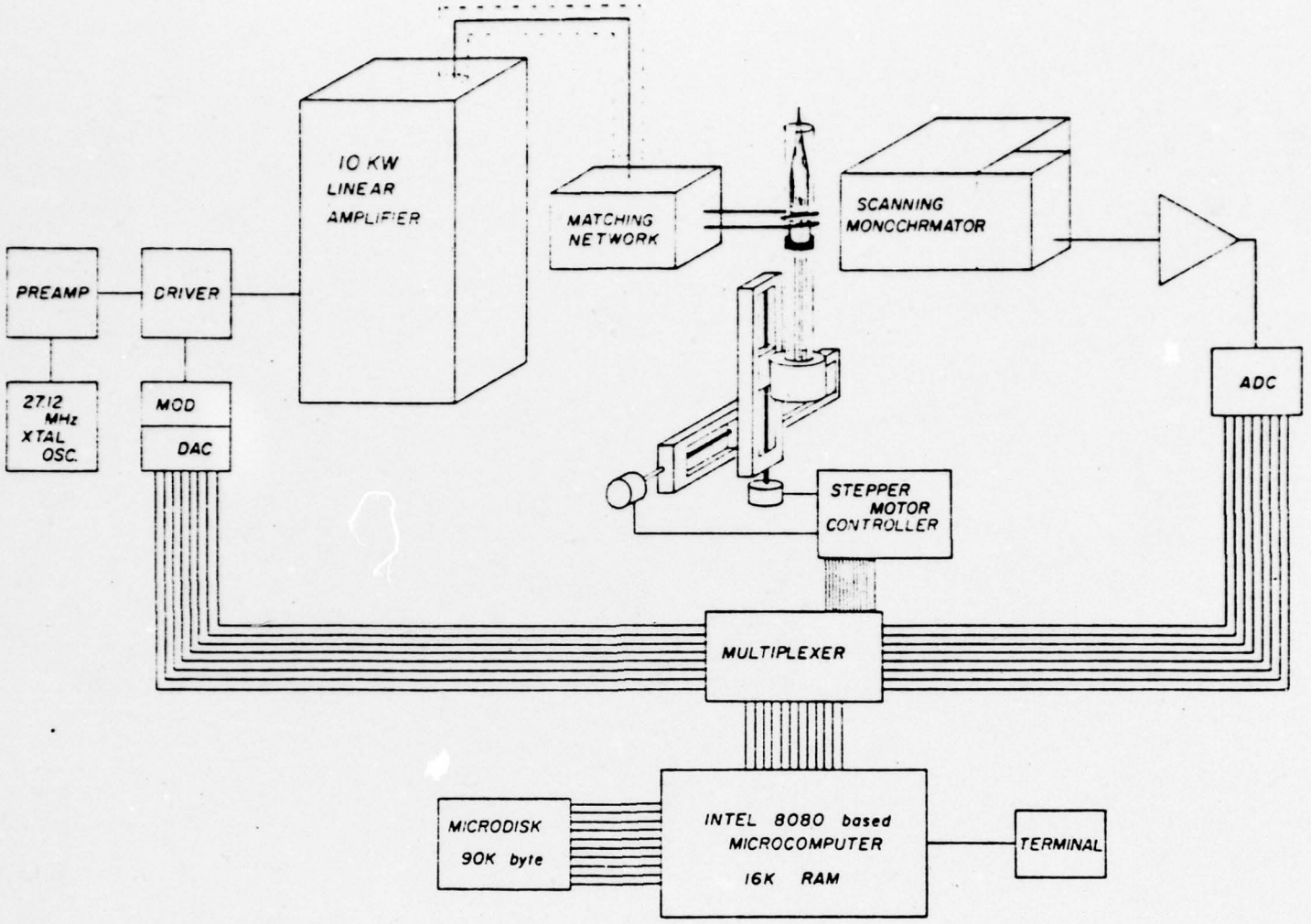
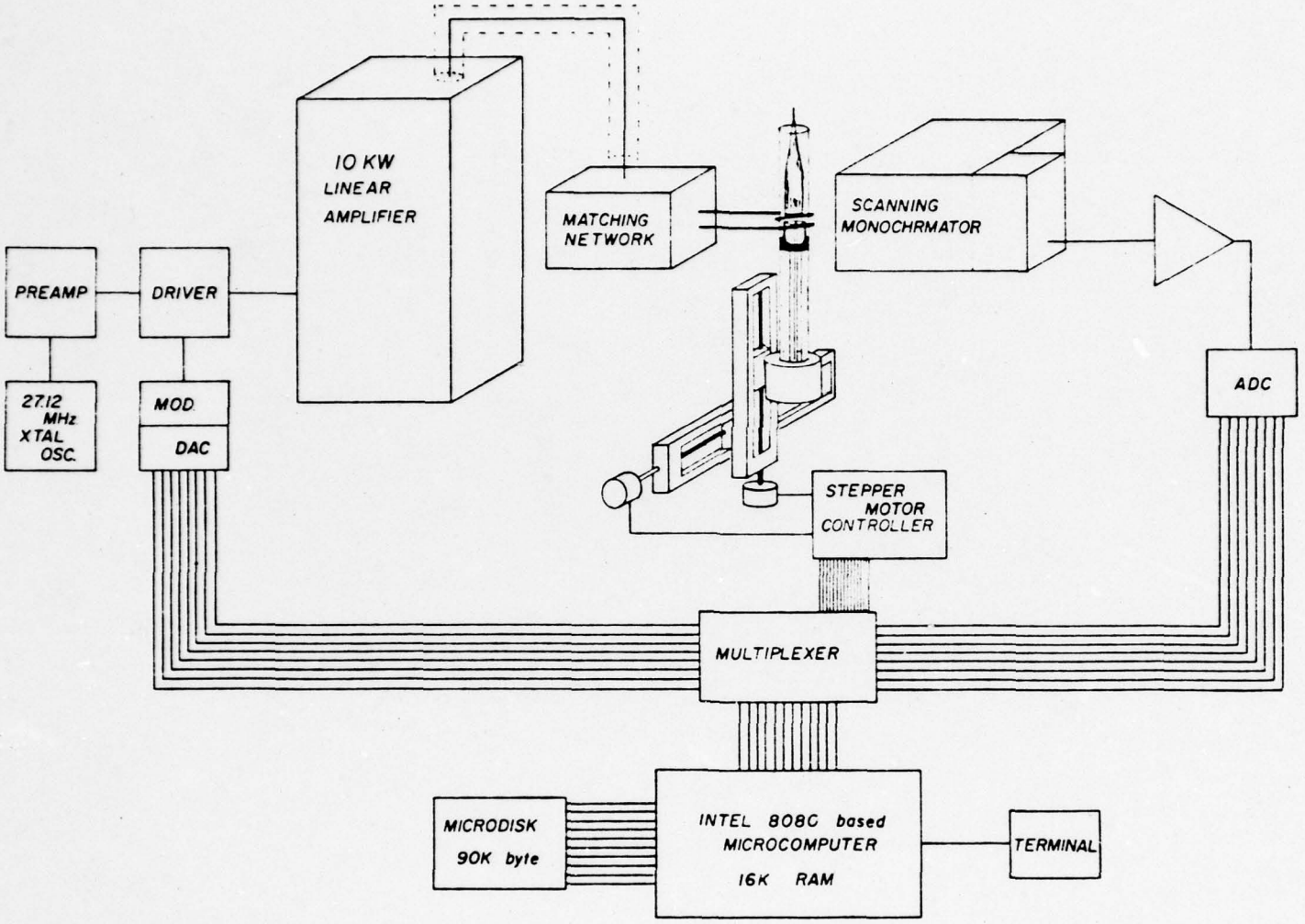


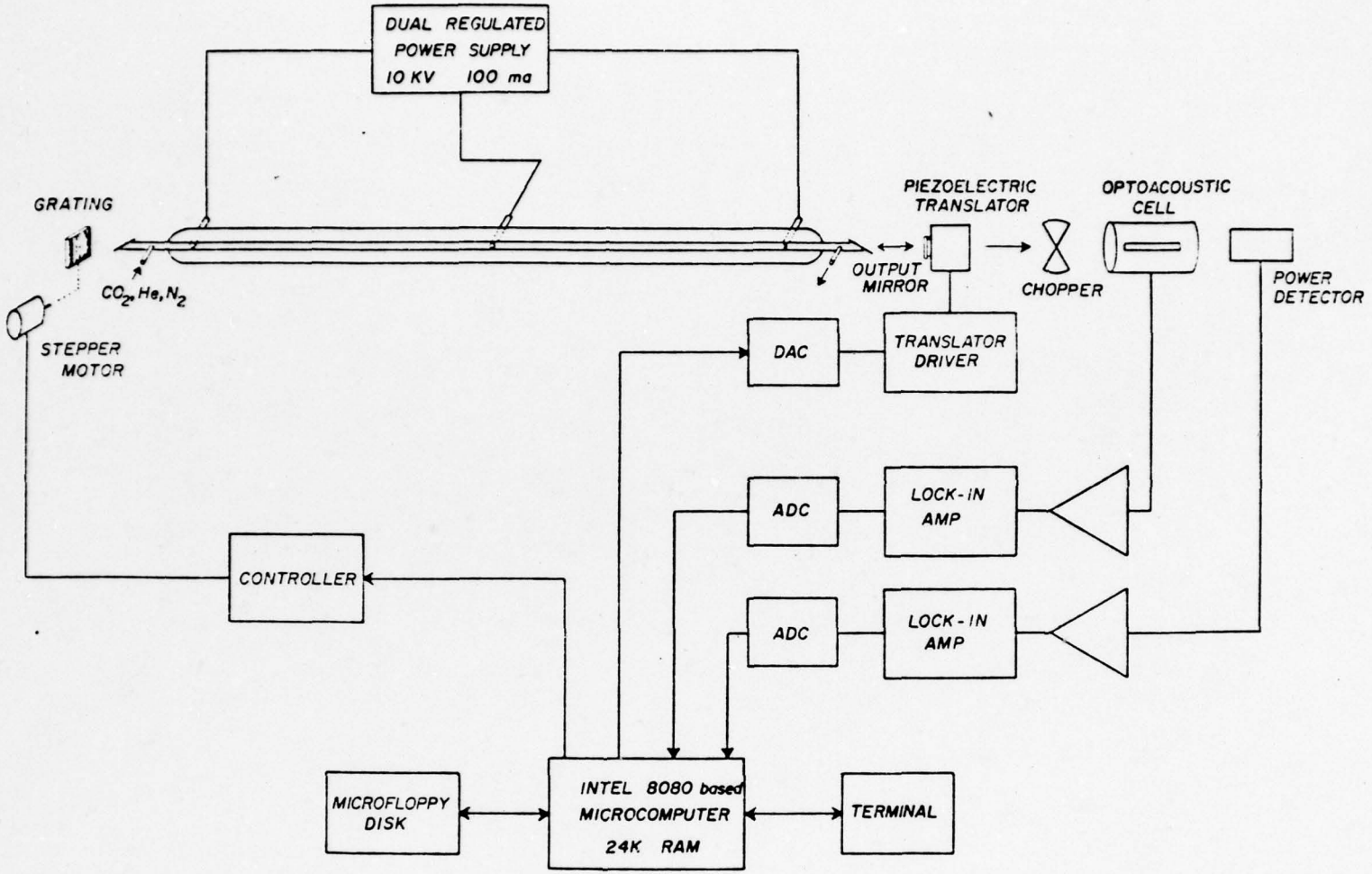
Figure 8. A schematic of the inductively coupled plasma emission spectrometer in which the microcomputer is used to control radio frequency power and 'flame' positioning as well as to monitor light intensity.

Figure Captions

Figure 7. The optoacoustic experiment in which a microcomputer is used to control laser wavelength and to monitor laser power and optoacoustic signal.

Figure 8. A schematic of the inductively coupled plasma emission spectrometer in which the microcomputer is used to control radio frequency power and 'flame' positioning as well as to monitor light intensity.


10 KW
LINEAR
AMPLIFIER
MATCHING
NETWORK
SCANNING
MONOCHRMATOR
PREAMP
DRIVER
27.12
MHz
XTAL
OSC.
MOD.
DAC
ADC
STEPPER
MOTOR
CONTROLLER
MULTIPLEXER
MICRODISK
90K byte
INTEL 808C based
MICROCOMPUTER
16K RAM
TERMINAL


DUAL REGULATED
POWER SUPPLY
10 KV 100 ma
GRATING
CO_2, He, N_2
STEPPER
MOTOR
PIEZOELECTRIC
TRANSLATOR
OPTOACOUSTIC
CELL
OUTPUT
MIRROR
CHOPPER
POWER
DETECTOR
DAC
TRANSLATOR
DRIVER
ADC
LOCK-IN
AMP
ADC
LOCK-IN
AMP
CONTROLLER
MICROFLOPPY
DISK
INTEL 8080 based
MICROCOMPUTER
24K RAM
TERMINAL

472:GAN:716:tam
78u472-608

TECHNICAL REPORT DISTRIBUTION LIST, GEN

	No. Copies
Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217 Attn: Code 472	2
ONR Branch Office 536 S. Clark Street Chicago, Illinois 60605 Attn: Dr. George Sandoz	1
ONR Branch Office 715 Broadway New York, New York 10003 Attn: Scientific Dept.	1
ONR Branch Office 1030 East Green Street Pasadena, California 91106 Attn: Dr. R. J. Marcus	1
ONR Area Office One Hallidie Plaza, Suite 601 San Francisco, California 94102 Attn: Dr. P. A. Miller	1
ONR Branch Office Building 114, Section D 666 Summer Street Boston, Massachusetts 02210 Attn: Dr. L. H. Peebles	1
Director, Naval Research Laboratory Washington, D.C. 20390 Attn: Code 6100	1
The Assistant Secretary of the Navy (R,E&S) Department of the Navy Room 4E736, Pentagon Washington, D.C. 20350	1
Commander, Naval Air Systems Command Department of the Navy Washington, D.C. 20360 Attn: Code 310C (H. Rosenwasser)	1
Defense Documentation Center Building 5, Cameron Station Alexandria, Virginia 22314	12
U.S. Army Research Office P.O. Box 1211 Research Triangle Park, N.C. 27709 Attn: CRD-AA-IP	1
Naval Ocean Systems Center San Diego, California 92152 Attn: Mr. Joe McCartney	1
Naval Weapons Center China Lake, California 93555 Attn: Dr. A. B. Amster Chemistry Division	1
Naval Civil Engineering Laboratory Port Hueneme, California 93401 Attn: Dr. R. W. Drisko	1
Professor K. E. Woehler Department of Physics & Chemistry Naval Postgraduate School Monterey, California 93940	1
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, D.C. 20380	1
Office of Naval Research 800 N. Quincy Street Arlington, Virginia 22217 Attn: Dr. Richard S. Miller	1
Naval Ship Research and Development Center Annapolis, Maryland 21401 Attn: Dr. G. Bosmajian Applied Chemistry Division	1
Naval Ocean Systems Center San Diego, California 91232 Attn: Dr. S. Yamamoto, Marine Sciences Division	1

Encl

472:GAN:716:tam
78u472-608

TECHNICAL REPORT DISTRIBUTION LIST, 051C

	No. Copies
Dr. M. B. Denton University of Arizona Department of Chemistry Tucson, Arizona 85721	1
Dr. R. A. Osteryoung Colorado State University Department of Chemistry Fort Collins, Colorado 80521	1
Dr. B. R. Kowalski University of Washington Department of Chemistry Seattle, Washington 98105	1
Dr. S. P. Perone Purdue University Department of Chemistry Lafayette, Indiana 47907	1
Dr. D. L. Venezky Naval Research Laboratory Code 6130 Washington, D.C. 20375	1
Dr. H. Freiser University of Arizona Department of Chemistry Tuscon, Arizona 85721	
Dr. Fred Saalfeld Naval Research Laboratory Code 6110 Washington, D.C. 20375	1
Dr. E. Chernoff Massachusetts Institute of Technology Department of Mathematics Cambridge, Massachusetts 02139	1
Dr. K. Wilson University of California, San Diego Department of Chemistry La Jolla, California	1
Dr. A. Zirino Naval Undersea Center San Diego, California 92132	1
Dr. John Duffin United States Naval Postgraduate School Monterey, California 93940	1
Dr. C. M. Hieftje Department of Chemistry Indiana University Bloomington, Indiana 47401	1
Dr. Victor L. Rehn Naval Weapons Center Code 3813 China Lake, California 93555	1
Dr. Christie G. Enke Michigan State University Department of Chemistry East Lansing, Michigan 48824	1
Dr. Kent Eisentraut, MBT Air Force Materials Laboratory Wright-Patterson AFB, Ohio 45433	1
Walter G. Cox, Code 3632 Naval Underwater Systems Center Building 148 Newport, Rhode Island 02840	1